5-2001

A Data Layout Descriptor Language (LADEL).

Ashfaq Ahmed Jeelani
East Tennessee State University

A Data Layout Descriptor Language (LADEL)

A thesis
presented to
the faculty of the Department of Computer and Information Science
East Tennessee State University

In partial fulfillment
of the requirements for the degree
Master of Science in Computer Science

---

by
Ashfaq A Jeelani
May 2001

Dr. Phillip Edward Pfeiffer IV, Chair
Dr. Donald Sanderson
Dr. Terry Countermine

Keywords: block I/O, data layout, dynamic data layout, network I/O,
file I/O, structured data

CONTENTS

LIST OF FIGURES

FIGURE Page

FIGURE Page

ABSTRACT

A Data Layout Descriptor Language (LADEL)

By

Ashfaq A Jeelani

To transfer data between devices and main memory, standard C block I/O interfaces use block buffers of type char. C++ programs that perform block I/O commonly use typecasting to move data between structures and block buffers. The subject of this thesis, the layout description language (LADEL), represents a high-level solution to the problem of block buffer management. LADEL provides operators that hide the casting ordinarily required to pack and to unpack buffers and guard against overflow of the virtual fields. LADEL also allows a programmer to dynamically define a structured view of a block buffer's contents. This view includes the use of variable length field specifiers that supports the development of a general specification for an I/O block that optimizes the use of preset buffers. The need for optimizing buffer use arises in file processing algorithms that perform optimally when I/O buffers are filled to capacity. Packing a buffer to capacity can require reasonably complex C++ code. LADEL can be used to reduce this complexity to considerable extent. C++ programs written using LADEL are less complex, easy to maintain, and easier to read than equivalent programs written without LADEL. This increase in maintainability is achieved at a cost of approximately 11 % additional time in comparison to programs that use casting to manipulate block buffer data.

CHAPTER 1

INTRODUCTION

1.1) Statement of Problem

Block I/O is an important tool for improving the performance of network and file-system applications. Sending data in chunks, rather than as individual bytes, can reduce the processing and latency overheads incurred by such applications. To do a block I/O transfer, the sender first packs the data into a single contiguous block of bytes, known as a *buffer*. The buffer is then sent as a whole to a device's driver, and sent as a unit, for eventual unpacking and processing by a second application.

This thesis considers the problem of how to manage I/O buffers in the C++ programming language. The main motivation for this research is to find ways of extending the C++ language to make buffer management code easier to write and maintain. C++ is generally regarded as a high-level language: one that encourages programmers to use carefully crafted data types and classes to structure data. Standard C++ block I/O methods, however, use block buffers of type char, rather than structs and classes, to transfer data between devices and main memory. These methods include file I/O methods like fstream::read() and fstream::write(), and network I/O methods like Winsock's send() and recv() methods. Using buffers of type char to support I/O made historic sense, in the context of C++'s evolution from C. Using I/O buffers of type char also simplifies the design of the C/C++ standard library, by allowing the library to provide just one interface for each type of block I/O routine. But the decision does shift the burden of

packing and unpacking buffers to the programmer.

C++'s reliance on char-oriented block I/O has encouraged the use of low-level code for C++ buffer management. One common strategy for manipulating buffers in C++ involves the use of C-style block move operators, along with typecasting, to move data between structs and block buffers. Each of a struct's constituent fields is first typecast to char *, and then moved, a byte at a time, into the I/O buffer. As a structure's complexity increases, the complexity of the code for manually packing the structs increases. Also, the very use of typecasting makes application-level code difficult to read and maintain.

A second strategy for buffer manipulation involves typecasting the whole struct as char *. Advocates of "structure-casting" argue that the pointer obtained from typecasting an entire structure will reference a block of memory containing the structure's data. Unfortunately, the C++ language specification does not guarantee that a structure's fields will be stored contiguously or even mapped into memory in consistent, compiler-invariant ways. So the C++ programs that use this strategy may face problems with portability, correctness, or both.

A third strategy for buffer manipulation in C++ is to stream data between classes and block buffers, using overloaded class operators like "<<" and ">>". Classes that support stream-based buffer packing and unpacking are commonly referred to as streamable classes. In such classes, the overloaded "<<" and ">>" operators transfer data between a class's internal variables and a target buffer. This strategy, though cleaner than the first two strategies, has problems of its own. Streaming operators are typically coded in ways that make them inflexible and unsafe. These operators typically force programmers to transfer data en masse between a buffer and an object,

and do not support direct buffer manipulation on a field-by-field basis. They force programmers to specify the layout of a buffer at compile time, and do not support the run-time determination of a buffer's contents. Finally, they are typically coded without checks for buffer underflow or overflow, arising from accidental misuse of the operators.

A final limitation of standard strategies for C++ buffer management is the lack of any support in the C++ language for automatically resizing a data structure to fill an available buffer. Here, the concern is supporting data structures like B-trees that, for optimum performance, should be expanded to fill whatever blocksize is "natural" for the underlying medium.

The outcome of this research is a specialized language for C++ programming that streamlines buffer management: the Layout Description Language, or LADEL.

1.2) Research Outcomes

LADEL is a "little" language that augments C++ with operators that hide the casting ordinarily required to pack and to unpack buffers. The language provides C++ programmers with a high-level, precise, and flexible language for accessing data written to and read from block I/O devices. The following is a list of features that this language, LADEL, provides:

- C-like declarations for specifying buffer layout, including layouts that support nested structs with named, typed fields.

- Support for dynamic buffer layout: i.e., the ability to distribute space in buffer that may be available at run time, beyond that buffer's minimum requirements to function. Support for dynamic buffer layout involves the following capabilities:
  1. The ability to specify minimum size requirement for each field in buffer.
  2. The ability to specify additional space to allocate for each field in buffer. Three possible ways for specifying additional space have been implemented:
     - as a specification of the minimum and maximum number of bytes per field;
     - as a function of the overall size of the buffer;
     - as a function of the amount of space remaining after a structure's minimum allocation has been met.
  3. The ability to determine, after an allocation is complete, the amount of storage allocated to each field.

- Support for operator-based buffer manipulation, including the following:
  1. Support for selection-operator-based field access.
  2. Support for safe, stream-based manipulation of individual fields, including checks for overflow and underflow on a per-field basis.

- Support for buffer manipulation with no language-specific extensions to C++. To meet this goal, LADEL was implemented as a language within C++, using a class whose constructor processes a layout specification in string format. No additions to the C++ grammar were required.

1.3) LADEL: An Introductory Example

The example specification shown in Figure 1.1 illustrates the feel and use of the LADEL language.

```
string   mySpecification  =
      string( " struct                          ") +
      string( " {                               ") +
      string( "   (int) field1;                 ") +
      string( "   struct                        ") +
      string( "   {                             ") +
      string( "           (char*5) field21;     ") +
      string( "           (float*3) field22;    ") +
      string( "   } field2; "                   ") +
      string ("}TopLevelSpec                    ");

   char  sourceBuf[128];

   BufferManagementClass
       BufferWrapper(sourceBuf, sizeof(sourceBuf), mySpecification);
```

Figure 1.1
Example LADEL Specification

The BufferManagementClass constructor, in effect, acts as the LADEL compiler. The constructor transforms the layout specification named mySpecification into a set of objects that support structured accesses to sourceBuf. Five such *buffer management objects* (BMOs) are generated from the declaration for BufferWrapper:

- The first object is an anonymous, top-level buffer BMO. This top-level object, which supports streaming into the buffer as a whole, references a contiguous block of storage that contains exactly sizeof(int) + 5*sizeof(char) bytes + 3*sizeof(float). This top-level object, when referenced with a selection operator, can return references to one of two second-level buffer management objects.
- The second object, a second-level BMO, is associated with the name "field1". This second

object references a contiguous, one-integer-long sub-block at the head of the underlying block I/O buffer.

- The third object, a second second-level BMO, is associated with the name "field2". This third object references a contiguous, five-character long + three-float-long sub-block that starts at the fifth byte position in the underlying buffer assuming that the size of integer is 4. This object is streamable and selectable. This object, when referenced with a selection operator, can return references to one of two third-level buffer management objects.
- The fourth object, a third-level BMO, is associated with the name "field21" This object references a contiguous, five-char-long sub-block that starts at the fourth byte position in the underlying buffer.
- The remaining object, a third-level BMO, is associated with the name "field22". It references a three-float-long sub-block that starts immediately after the last byte in field21.

LADEL provides access to individual fields through selection and streaming operators. The selection operators include ^ (select by name) and [] (select by index). The streaming operators include << (insert into buffer) and >> (extract from buffer). For example, a statement like

BufferWrapper^"field1" << 23456;

streams the value 23456 into field1 of the structure shown in Figure 1.1. A detailed discussion of the operators supported by LADEL is provided in section 3.3 and section 3.4 of Chapter 3.

Figure 1.1 is meant to suggest how LADEL provides C++ programmers with high-level access to data in buffers. LADEL provides data abstraction at a higher level than casting and streaming, without restricting how programmers structure data. LADEL also allows C++ programmers to

specify how buffers are structured at run time. LADEL BMOs allow programs to access a buffer's logical subfields and to do so with built-in error checks that guard against underflow, overflow, and other forms of improper field access.

As a part of this work, tests were run to determine how using LADEL would impact program performance. In one set of these tests one record of data was manually packed 500,000 times into a buffer. In a related set of tests data were laid out and packed 500,000 times using LADEL. When these tests were performed using a simple data layout the time taken by the program using LADEL was twice the time taken by the program that did not use LADEL. When the same tests were performed using a very complex data layout with three level of nesting the time taken by the program using LADEL was 10 times the time taken by program not using LADEL. It was determined that this degradation in performance was due to increasing number of function calls required for selection operations as we go deeper in the object hierarchy.

The same tests, however, were then rerun with hand-optimized code. The LADEL selection operation was applied once, and the object pointers returned by this operation were saved. The LADEL-based test code then manipulated the buffers an additional 499,999 times, using the saved pointers from the initial selection operation. Eliminating redundant selection operations substantially improved LADEL performance: the hand-optimized LADEL code yielded a performance degradation of 11% as compared to programs that performed typecasting-based buffer manipulation.

1.4) Overview of This Thesis

The balance of this document consists of four chapters. Chapter two discusses background issues connected with the development of LADEL. This chapter considers the role of block oriented I/O in mainstream programming. It discusses standard C++ strategies for doing block I/O in more detail. It then concludes with a discussion of Microsoft's second order file system (SOFS), one of the ideas that inspired this work. Chapter two also describes how LADEL would support the development of an SOFS-like program and describes the problems associated with heterogeneous data transfer over a network.

Chapter three gives a detail explanation of LADEL with a comprehensive set of examples. It provides a formal grammar for LADEL and describes the operators supported by LADEL. Chapter 3 also documents LADEL's key class, the BufferManagementClass.

Chapter four compares the performance of programs written using LADEL with programs written without using LADEL. Chapter four also demonstrates the ease of use of LADEL with two specific code examples.

Conclusions along with suggestions for improvements to LADEL are given in chapter five.

CHAPTER 2

BLOCK I/O

Applications that communicate using stream based protocols typically use block-oriented rather than byte-oriented send and receive. Block-oriented send and receive results in much better performance than byte-at-a-time I/O. Similarly, applications that share data using disk files use block-oriented reads and writes. This chapter discusses the importance of block I/O, along with a data management technology that has similarities to the ideas developed in this thesis.

Section 2.1 discusses the importance of block I/O in programming applications. The usefulness of dynamic data structure creation for some types of commercial applications is also considered. These commercial applications are developed in two parallel phases. In the first phase the actual functionality of the application is developed. In the second phase the data layout of the application is configured.

Section 2.2 discusses the role of block oriented I/O in mainstream C++ programming. Common casting-based and streaming-based strategies for manipulating block buffers and problems with those strategies are also discussed.

Section 2.3 discusses Microsoft's second order file system (SOFS). The goals of a second order file system (SOFS) are similar to those that motivated the development of LADEL. An SOFS is a file system within a file system. An SOFS provides applications with a structured storage where the applications can store different types of data as different objects within a file. An

SOFS then provides high-level direct access to those objects within the file. LADEL provides C++ programmers with a way to define structured data in physical memory and then provides high-level direct access to individual data elements within the structure.

2.1) Importance of Block I/O in Programming Applications

Network applications that need to send and receive data across a network perform block-oriented rather than byte-oriented I/O. For example if an application wants to send 80 KB of data across a network, and sends it 1 byte at a time, it would require 80 thousand send operations to transfer the 80 KB of data. Sending the same data in blocks of 8 KB would require only 10 send operations. As the latency associated with send operations is quite considerable, reducing the number of send operations results in improved performance. So network applications usually pack a large chunk of data into a block buffer before transferring that data across a network.

Similarly, applications that share data using disk files perform block-oriented reads and writes. Block-oriented reads and writes are done to take advantage of the block-oriented data storage mechanism of the disk. Generally a disk is divided into sectors and sectors are divided into blocks. When data are accessed as blocks, one positioning of a read/write head allows multiple bytes of data to be accessed. Accessing the same data on byte-by-byte basis would potentially require as many read/write head positioning as there are bytes to be accessed. As the mechanical movement of the read/write head is the most expensive operation in the whole process of reading and writing data from disk files, performing block-oriented I/O yields better performance than byte-oriented I/O.

Network I/O is an integral part of applications that are developed using the client-server paradigm. In the client-server paradigm most of the functionality is developed in the client and the data is stored in server. Then the data are transferred from the server to its clients at the clients' requests. This transfer is done in blocks of data as discussed above. The data in the block buffer received from server are unstructured, and must be structured by the clients for each request. This structuring requires very low level coding, and makes application programming difficult. A high-level language construct that supports the dynamic structuring of buffered data would allow the application programmer to focus more on developing the actual functionality. This same concern applies to applications that read their data from disk files. They read data in blocks and then structure these data in memory. Just how such an application must structure its data may depend on a file's actual content. For example, the precise layout of data in a file with self-describing data may not even be predictable at compile time. A high-level capability for defining a structured view of a buffer at run-time would simply the task of block I/O management for applications that manipulate such files.

An important class of applications that would benefit from dynamic buffer specifications is those applications whose functionality and user interface (UI) vary, depending on information discovered at run-time. Consider, for example, an application that needs to show different UIs for different users. The application developer needs, as part of this application, to write some logic where he can read the layout of UI for different users and generate the UI. As each UI will have a different set of data, the programmer needs to create some sort of dynamic data structure to hold each UI's data. C++ does not support the dynamic specification of data structures directly. To achieve his goal, the programmer must create some sort of data structure, like a

linked list, to hold the data for the UI. Linked list creation and management is a low level operation and its coding is error prone.

The commercial application market features many applications available that are self-configuring, based on startup files or other metadata.[1] These self-configuring applications are developed in two parallel phases. In the first phase the actual functionality is developed. In the second phase the data layout for each individual UI is configured. This data layout is configured using tools that write the layout into some file or database. The application developed in the first phase reads the layout from this file or database and displays the data accordingly. The coding for such an application is very complex and requires very complex mechanisms for managing the data dynamically. If some sort of dynamic data structure specification facility were provided to these commercial applications, the development and testing time could be reduced significantly. Also the resulting code would be less complex and easier to read and to maintain.

2.2) Role of Block-Oriented I/O in Mainstream C++ Programming

C++ supports a variety of useful high-level features for software development, including support for strong typing, classes, inheritance, and exception handling. C++, however, lacks a set of high-level language constructs for positioning data in physical memory. This ability to position data in memory is important for inter-program communication via unstructured media. Programs that exchange data via unstructured media need guarantees about how I/O positions data in physical memory to synchronize data accesses. Examples of such programs include

[1] This discussion of configurable applications is highly nonspecific, for reasons of confidentiality.

network applications that communicate via stream-based protocols and applications that use stream-based and block-based files to share data. Because I/O buffers are typically heterogeneous data structures, what is wanted is a set of guarantees about how heterogeneous objects like structs are mapped to physical memory. These layout guarantees *could* be provided by the language specification, or by the specification for the language's run-time system or by the implementation itself—but they *must* be present for correct operation.

The C++ language standard does not provide the desired guarantees about how data in structs are mapped to memory [Koenig]. Standard C++ input and output methods—methods like fstream::read(), fstream::write(), and Winsock's send() and recv() methods—operate on buffers of type char rather than structs. This lack of support for struct-based I/O forces the programmer who is concerned about the integrity of data transfers to manually pack heterogeneous data into buffers of type char, and then to unpack the buffers at the receiver. The resulting code is neither easy to read nor maintain. The C memcpy() function, which moves data between sets of locations, can be used to simplify packing. memcpy(), however, is still a low-level primitive, and yields low-level code (cf. Figure 4.3).

At least two alternatives to character-buffer-based I/O have been proposed. Some authorities suggest that a struct be passed to a method like fstream::write() by casting the entire struct as an object of type char* [Uckan]. This approach to doing I/O with structs, however, is unsafe, because the C++ standard fails to specify how the individual components of classes and structs are to be positioned in physical memory[2]. The other alternative, which is used with classes, is to

[2] Stroustrup has stated that C++ compilers are required to lay out the structs contiguously in memory [Stroustrup]. This constraint, however, is not an explicit part of the standard.

make a class *streamable*. In a streamable class, overloaded versions of the "<<" and ">>" operators are provided that transfer data between a class's internal variables and a stream. Parrington[Parr] discusses a similar use of the << and >> operators in the context of buffer management for remote procedure call. Even though using streamable classes for doing buffer I/O is better than using casting, there are two other desirable features which streamable classes lack:

- Streamable classes do not allow the programmers the ability to treat a block as a hierarchical object, made up of sub-blocks that can be streamed individually. This feature would be helpful for developing applications that pack multiple logical objects into a single physical data structure, like a second-order file system (cf. section 2.3). Providing direct access to different types of data stored within a single file makes the job of writing applications that require storing structured data, easier. Treating different types of data as objects within a file reduces the complexity of file sharing semantics.
- Streamable classes do not give programmers the ability to specify a buffer's structure dynamically, a feature which is required in situations where the size of device's block buffers are not known at compile time. C++ does not allow the number of records or size of individual records to expand a feature that if provided can take advantage of an I/O subsystem's underlying block size.

2.3) Microsoft's Second Order File System

Microsoft implemented a 'second order file system' (SOFS) as part of its COM (Component Object Model) software package. COM is designed to promote *software interoperability*: that is, to allow arbitrary applications running on arbitrary systems to share objects. COM defines mechanisms and interfaces that allow applications to connect to each other as software objects. COM's storage-related interfaces are collectively called Persistent Storage or Structured Storage.

COM implements persistent storage as an SOFS. In COM's implementation of persistent storage, a single file entity is treated as a structured collection of *storages* and s*treams*. Storages and streams act like directories and files respectively. A Stream object in COM is the conceptual equivalent of a single disk file and a Storage object is the conceptual equivalent of a directory. Streams hold data, are associated with access rights, and are accessed with a single seek pointer. Storages, which are also associated with access rights, contain arbitrary numbers streams and sub-storages. Storages and streams are implemented in standard formats and can be shared between processes.

The following example, taken from [Microsoft], illustrates the use of COM to simplify the development of an application that stores highly structured data. The application in question, a diary program, allows a user to associate entries that represent days with objects that represent daily events. These objects are permitted to contain different kinds of information: text objects for textual information, bit maps for newspaper images, and so forth [Microsoft]. The resulting network of objects could easily contain multiple levels of indirection and cross-referencing, (cf. Figure 2.1, reproduced from [Microsoft]). COM simplifies the development of the diary application by

allowing a developer to create a hierarchy of storages that organize events according to the flow of time. This hierarchy would consist of streams representing specific events, contained in storages for individual days, contained in storages for individual months, grouped into storages for specific years (cf. Figure 2.2, reproduced from [Microsoft]). COM also provides a set of file-system-like APIs that map these objects and storages into a single, flat file automatically, on the user's behalf. The problem of expanding information in objects is solved as the object itself expands the stream in its control. The implementation of the "file system within a file" determines where to store information on the diary application's behalf, making the application easier to code.

Standard file system primitives, like the ones found in UNIX and other major operating systems, allow users to organize data in a related set of files and directories, in a way that is logically similar to how an SOFS like COM stores data in different objects within a file. The proponents of SOFS, however, argue that SOFSes are better for managing certain kinds of data than directories. Directory structures, according to this argument, are useful for loosely coupled system of files, while SOFSes are useful for managing tightly coupled systems of data objects. Files and directories consume system resources, and are generally awkward for maintaining a dataset that consists of a lot of small, interrelated objects. Hence SOFS is desirable for applications that need to store highly structured data within a file.

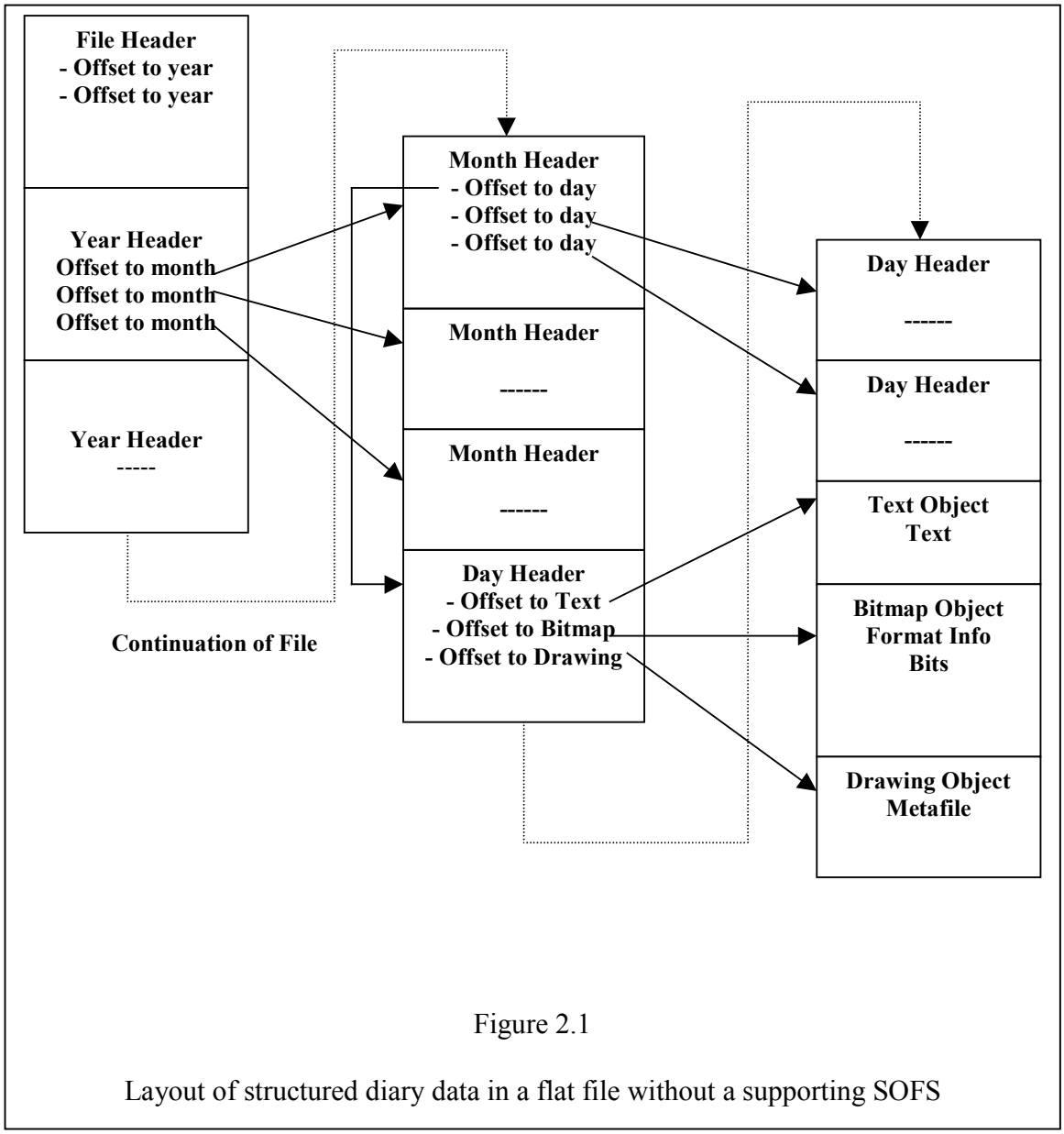

Figure 2.1

Layout of structured diary data in a flat file without a supporting SOFS

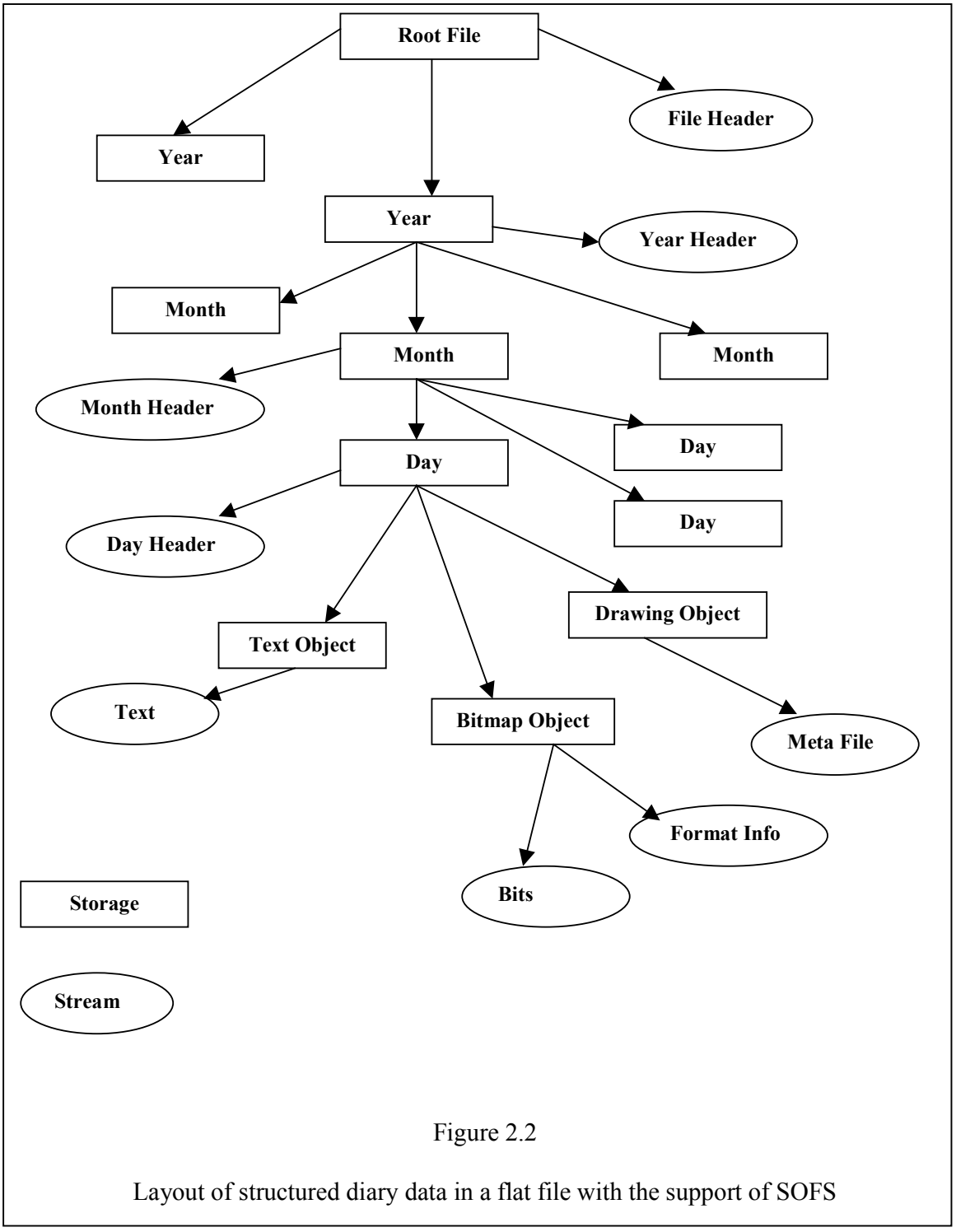

Figure 2.2

Layout of structured diary data in a flat file with the support of SOFS

CHAPTER 3

LADEL

The Layout Descriptor Language, or LADEL, allows programs to manipulate blocks of *raw* (i.e., unstructured) storage as hierarchical data structures. This structuring of raw storage takes place at run-time, using two kinds of constructs:

- a set of declarators, which allow a programmer to define a structured view of a block of raw storage;
- a buffer management object (BMO), which "realizes" a specific view of storage, relative to a specific raw storage buffer.

Other LADEL operators then allow programs to query BMOs for status information and to manipulate the raw storage associated with a BMO as a hierarchy of named, typed streams.

The LADEL language was designed for use in a standard C++ environment. The balance of this chapter describes LADEL syntax and implementation and illustrates its use with a series of examples. Section 3.1 describes the LADEL language. Topics include basic LADEL syntax; selection and streaming operators; nameless and variable-length fields; LADEL arrays; and the BLOCKSIZE and SURPLUS keywords. Section 3.2 specifies LADEL's formal grammar. Section 3.3 gives an overview of the operators supported by LADEL. Section 3.4 concludes with an overview of the LADEL Block Management Object.

3.1) The LADEL Language

The LADEL language was developed to provide C++ users with a high-level view of block-buffered I/O. LADEL simplifies the development of standard C++ code for buffer manipulation in two ways. The LADEL language allows programmers to stream typed data into and out of block buffers by referencing named fields whose sizes, types, and offsets are computed from user-supplied declarations. LADEL also simplifies the development of code that needs to adapt to buffers of varying size. The LADEL language allows users to specify, as a part of a field's definition, a range of sizes and/or repetition counts for that field. LADEL then matches the declaration to the supplied buffer at run-time, using a top-down "spare byte distribution" algorithm. Support for this feature was added to the language specifically for applications like B-trees, which should attempt to pack as many records into each buffer as the underlying disk I/O system will allow.

LADEL has been implemented as C++ class with a constructor that parses C++-like specifications for data layouts. A LADEL class, known as BufferManagementClass, generates objects that manage program access to an associated block buffer at run-time. The BufferManagementClass constructor accepts, as one of its arguments, a specification of a data layout in the form of a string. This string defines a view of storage that, roughly speaking, is comparable to the view determined by a C/C++ struct. The LADEL data layout string, in particular, defines a hierarchically structured, typed data object that, in turn, may consist of other structs, nested to an arbitrary depth. Each structure and substructure is referred to as a *field*. The BufferManagementClass constructor ultimately maps each field in the data layout to a range of

offsets in a user-specified block buffer. The BufferManagementClass constructor also instantiates one object of type BufferManagementClass for managing access to each field. These *buffer management objects* (BMOs) are themselves arranged in a hierarchy that is isomorphic to the hierarchy defined by the declaration.

Every BMO generated for an I/O buffer manages data operations on one region of the buffer. BMOs support the safe streaming of data between elementary C++ data types and a buffer's sub-blocks: range checking is done to ensure that streaming operations do not overrun block boundaries. BMOs also support selection of individual fields from the data layout.

The features and uses of the LADEL language are illustrated below, using a progressive series of examples.

3.1.1) Basic LADEL Specification Syntax

The layout specification in Figure 3.1 is a simple example, which shows how a data layout can be specified using LADEL. The syntax is very similar to C++ except for the size specifier, (int * 5), which specifies the space that a subregion in a block should occupy, in bytes. LADEL's BufferManagementClass constructor converts the layout specification in Figure 3.1 into the 5 BMOs shown in Figure 3.2. For example, the BMO labeled "field2" controls access to the eight-integer long sub-block that starts at the fourth byte position in sourceBuf. Through this BMO another two BMO's (for *field21* and for *field22* respectively) can be referenced, which control the access of the 3 integer-long field *field21* and the 5 integer-long field *field22*. The constructor

also gives the programmer precise control over how data are arranged in memory. In the layout specification given in Figure 3.1 *field1* will be positioned immediately before *field2*, and *field22* immediately after *field21*.

```
string   mySpecification  =
      string(“ struct                          ”) +
      string(“ {                               ”) +
      string(“    (char*3) field1;             ”) +
      string(“    struct                       ”) +
      string(“    {                            ”) +
      string(“          (int*5) field21;       ”) +
      string(“          (int*3) field22;       ”) +
      string(“    } field2; ”                  ”) +
      string (“}buffer                         ”);

char sourceBuf [128];

BufferManagementClass
      BufferWrapper (sourceBuf, sizeof (sourceBuf), mySpecification)
```

Figure 3.1
Simple LADEL specification with semantics

3.1.2) LADEL Selection and Streaming Operators

BMOs, once constructed, can be manipulated using four overloaded operators:

- The >> operator, which streams a data item from a BMO into its right operand.
- The << operator, which streams a data item from into its right operand into a BMO.
- The ^ operator, which selects a sub field from a BMO.
- The [] operator, which selects one of a set of related fields.

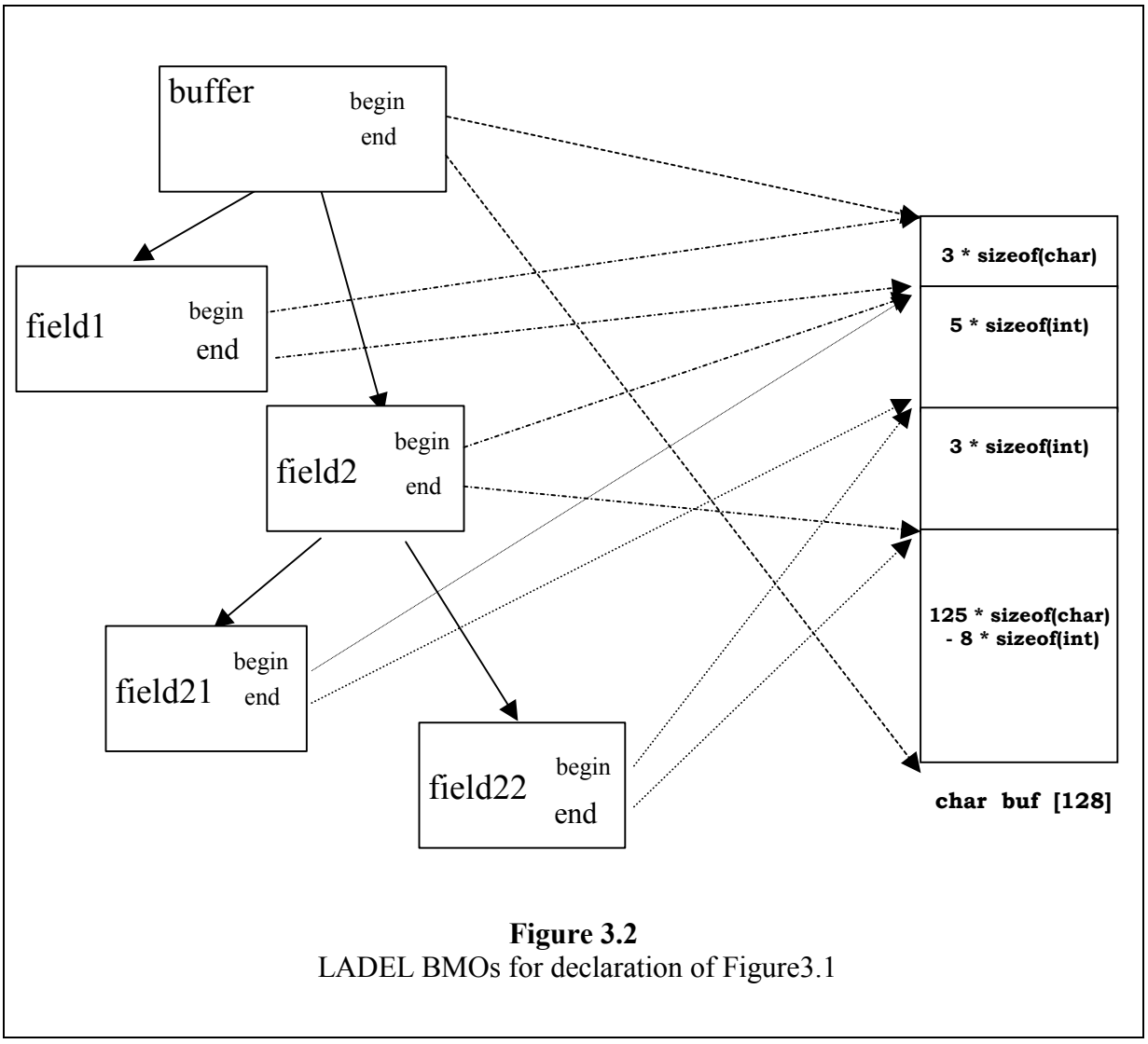

Figure 3.2
LADEL BMOs for declaration of Figure3.1

The >> and << operators use a "current position" cursor associated with each BMO. This object keeps track of the next byte to be written into or read from underlying buffer. The current position in a BMO can also be adjusted and inspected, using setpos() and getpos() methods, respectively. The selection and streaming operators raise exceptions when an unknown or anonymous field is selected, or when a streaming operator attempts to access locations beyond the boundaries of a sub field. The ^ operator was overloaded for use as a selection operation

because the standard selection operator, dot (.), cannot be overloaded in C++. The indexing operator [] is discussed further in section 3.1.5.

3.1.3) Nameless Fields

Nameless fields are fields that do not have any name by which they can be referred. Only their sizes are declared in the Data Layout. A nameless field in a LADEL specification cannot be selected, or manipulated directly. Nameless fields can be used for inter-field padding. Inter-field padding can be used to ensure data alignment in environments where alignment is required: e.g., on processors that lack alignment networks. One example of a nameless field is given in the layout specification in Figure 3.3, where the third field acts as a space holder.

```
string mySpecification =
            string("struct                    ") +
            string("{                         ") +
            string("    (char*3) field1;      ") +
            string("    (int*5) field2;       ") +
            string("    (1);                  ") +
            string("}                         ");
```

Figure 3.3
Example with nameless fields

3.1.4) Variable-length Fields

LADEL allows users to specify fields whose size can vary, relative to the amount of space present in a buffer at run-time. LADEL allows a programmer to specify the minimum number of bytes that should be reserved and also the number of bytes beyond the minimum that should be assigned to a field, if space is available in the underlying buffer. For example

(2,6) field1;

specifies that field1 should contain a minimum of 2 bytes and that a maximum of 8 bytes (i.e., 6 additional bytes beyond the first 2) can be assigned to field1.

The following series of examples shows how LADEL distributes the surplus space in the underlying buffer among different fields of the struct based on min/max values of the fields. The layout descriptor shown in Figure 3.4 specifies a family of storage assignments, where the actual assignment is dependent upon the size of the underlying buffer.

When the example in Figure 3.4 is executed with BUFFER_SIZE of less than 3, the BufferManagementClass constructor generates a "no assignment possible" exception.

```
string variableSpecification =
     string("struct                                          ") +
     string("{                                               ") +
     string( "    (1,1)field1;                               ") +
     string("     (2,3)                                      ") +
     string("     struct                                     ") +
     string("     {                                          ") +
     string("           (1,2) field21;                       ") +
     string("           (1,1) field22;                       ") +
     string("     }field2                                    ") +
     string("}buffer                                         ");


char  sourceBuf[BUFFER_SIZE];

BufferManagementClass
      BufferWrapper (sourceBuf, sizeof(sourceBuf),
                                          variableSpecification)
```

Figure 3.4
A layout descriptor with min/max values for each field

When the example in Figure 3.4 is executed with BUFFER_SIZE equal to 3, the declaration in Figure 3.4 is equivalent to the following, fixed-length declaration:

-. struct{(1)field1;{struct{(1)field21; (1)field22;}field2;} buffer;

As the minimum requirement for the layout is 3 all the fields will get their minimum requirements (field1 gets 1 byte, field21 gets 1 byte, and field22 gets one byte).

The memory layout for the declaration in Figure 3.4, relative to a buffer size of 3, is shown in Figure 3.5 below.

When the example in Figure 3.4 is executed with BUFFER_SIZE equal to 4, the declaration in Figure 3.4 is equivalent to the following, fixed-length declaration:

-. struct{(2)field1;{struct{(1)field21; (1)field22;}field2;} buffer;

As the minimum requirement for the layout is 3, there is one surplus byte. According to the rules of surplus storage distribution this byte goes to field1. The above declaration's memory layout is shown in Figure 3.6.

When the example in Figure 3.4 is executed with larger buffers, the declaration in Figure 3.4 is equivalent to the following, fixed-length declarations:

-. struct{(2)field1;{struct{(2)field21; (1)field22;}field2;} block; (BUFFER_SIZE = 5);

-. struct{(2)field1;{struct{(3)field21; (1)field22;}field2;} block; (BUFFER_SIZE = 6);

-. struct{(2)field1;{struct{(3)field21; (2)field22;}field2;} block; (BUFFER_SIZE >= 7)

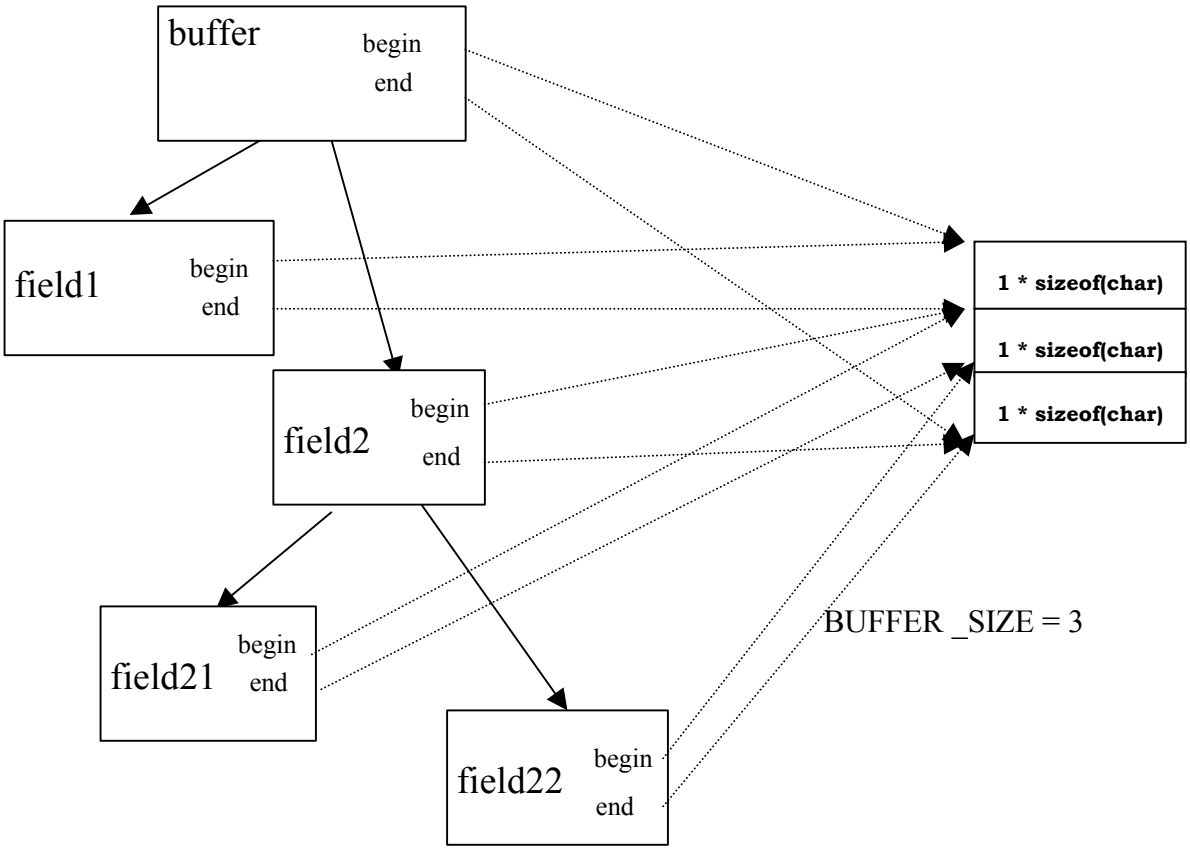

Figure 3.5
One possible memory layout for the layout descriptor of Figure 3.4

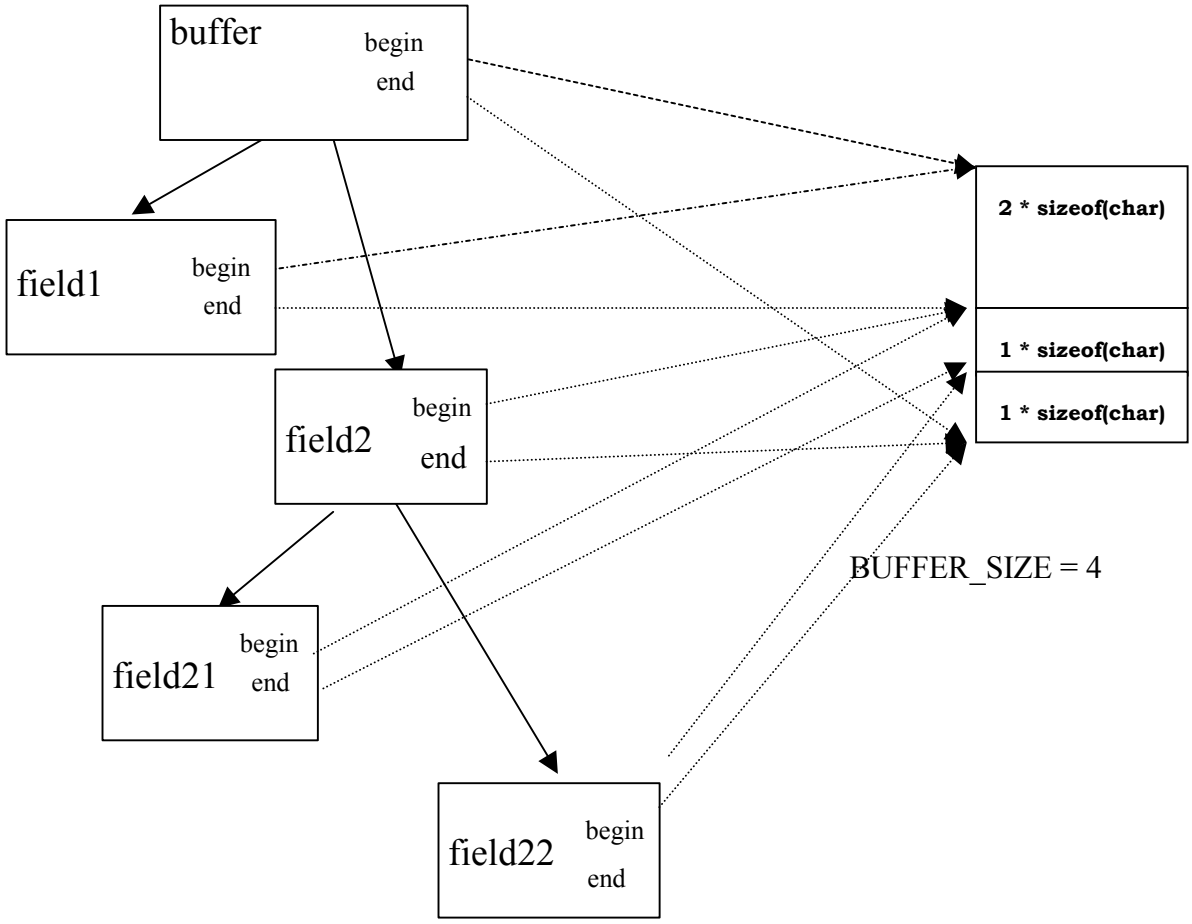

Figure 3.6
Another possible memory layout for the layout descriptor of Figure 3.4

In the above examples, LADEL first distributes available storage in a way that meets the minimum requirements of each field. LADEL distributes the surplus storage in a top-down, left-to-right manner: a field F's allotment of surplus storage is restricted to the amount of storage that its parent has obtained, and limited to the amount of storage that remains after the storage requests of F's predecessor siblings have been satisfied. In the above example, field1's maximum requirements would be satisfied from the surplus storage before any other field.

3.1.5) Array Specifier

An array specifier of N, when appended to a field name F, directs LADEL to generate N consecutive instances of the specified field, named F[0] … F[N-1] inclusive. The layout descriptor of Figure 3.7, for example, would generate the same layout as shown in Figure 3.8 if

```
string      variableSpecification =
     string("struct                                    ") +
     string("{                                         ") +
     string("     (2,4)                                ") +
     string("     struct                               ") +
     string("     {                                    ") +
     string("          (1,2)field11;                   ") +
     string("          (1,2) field12;                  ") +
     string("     }field1[2]                           ") +
     string("     (1,1)field2;                         ") +
     string("}buffer                                   ");

char  sourceBuf[BUFFER_SIZE];

BufferManagementClass
      BufferWrapper (sourceBuf, sizeof(sourceBuf),
                                    variableSpecification);
```

Figure 3.7
Example with arrays of structs in specification

integers were allowed as field names. The data layout of Figure 3.8 is depicted here just for explanatory purposes, as LADEL does not permit use of integers as field names.

The layout descriptor of 3.7 generates the equivalent of one of the following storage assignments, depending on the number of bytes in the underlying buffer:

```
string variableSpecification =
     string("struct                                          ") +
     string("{                                               ") +
     string("     (2,4)                                      ") +
     string("     struct                                     ") +
     string("     {                                          ") +
     string("           (1,2) field11;                       ") +
     string("            (1,2) field12;                      ") +
     string("       }0;                                      ") +
     string("     (2,4)                                      ") +
     string("     struct                                     ") +
     string("     {                                          ") +
     string("           (1,2)field11;                        ") +
     string("           (1,2)field12;                        ") +
     string("     }1;                                        ") +
     string( "(1,1)field2;                                   ") +
     string("}buffer                                         ");

char  sourceBuf[BUFFER_SIZE];

BufferManagementClass
            BufferWrapper (sourceBuf, sizeof(sourceBuf),
                                    variableSpecification);
```

Figure 3.8
Layout Equivalent to the Data Layout of Figure 3.7

-. "no assignment possible" exception (BUFFER_SIZE <= 4).

-. struct{struct{(1) field11,(1)field12}field1[2]; (1) field2 }buffer; (BUFFER_SIZE = 5);

-. struct{struct{(1) field11, (1) field12}field1[2]; (2) field2}buffer; (BUFFER_SIZE = 6);

-. struct{struct{(2) field11, (1) field12}field1[2]; (1) field2}buffer; (BUFFER_SIZE = 7);

-. struct{struct{(2) field11, (1) field12}field1[2]; (2) field2}buffer; (BUFFER_SIZE = 8);

-. struct{struct{(3) field11, (1) field12}field1[2]; (1) field2}buffer; (BUFFER_SIZE = 9);

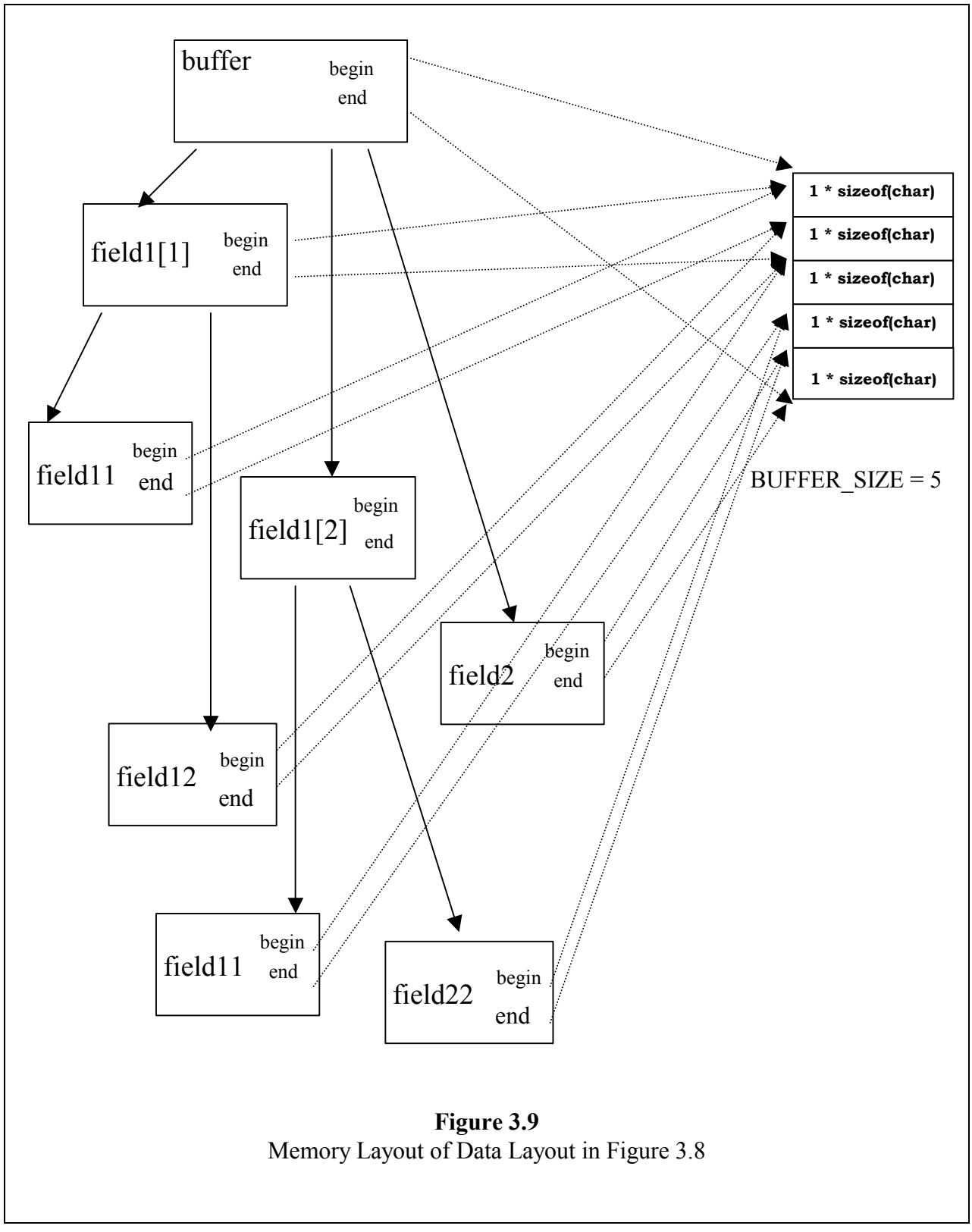

Figure 3.9
Memory Layout of Data Layout in Figure 3.8

-. struct{struct{(3) field11,(1) field12}field1[2];(2) field2}buffer; (BUFFER_SIZE = 10);

-. struct{struct{(3) field11,(2) field12}field1[2];(1) field2}buffer; (BUFFER_SIZE = 11);

-. struct{struct{(3) field11,(2) field12}field1[2];(2) field2}buffer; (BUFFER_SIZE = 12);

-. struct{struct{(3) field11,(3) field12}field1[2];(1) field2}buffer; (BUFFER_SIZE = 13);

-. struct{struct{(3) field11,(3) field12}field1[2];(2) field2}buffer; (BUFFER_SIZE ≥ 14);

The surplus storage distribution rules of example 4 apply to arrays, except that LADEL guarantees that the number of bytes allocated to each element of the array will be same.

3.1.6) The BLOCKSIZE Keyword

LADEL allows the use of special keyword BLOCKSIZE as a size qualifier that denotes the number of bytes in the underlying block buffer. The layout descriptor shown Figure 3.10, when presented with a BLOCKSIZE-byte-long block buffer, reserves the first ⌊BLOCKSIZE/2⌋ bytes for field1, and the remaining ⌊BLOCKSIZE/2⌋ bytes for field2.

```
string   mySpecification =
          string("struct                                    ") +
          string("{                                         ") +
          string("      (BLOCKSIZE/2)field1                 ") +
          string("      (BLOCKSIZE/2) field2;               ") +
          string("}buffer;                                  ");

char  sourceBuf[BLOCKSIZE];

BufferManagementClass
      BufferWrapper(sourceBuf, sizeof(sourceBuf), mySpecification);
```

Figure 3.10
Use of BLOCKSIZE keyword as a size qualifier an array

The BLOCKSIZE keyword may also be used to specify the number of elements in an array of storage. The layout descriptor shown in Figure 3.11 would generate the following set of storage assignments, relative to an n-byte-long buffer:

-. struct{struct{(3) field11;(2) field12;}field1[k];}buffer;

$$(n = 5k + j,\ k \geq 0;\ j = 0,1,2,3,4)$$

Intuitively, this declarator, when evaluated, allocates the first $5*\lfloor n/5 \rfloor$ bytes in sourceBuf to $\lfloor n/5 \rfloor$ records, named field1[0]…field[$\lfloor n/5 \rfloor$-1]. The final n mod 5 bytes in sourceBuf are not assigned to any record.

```
string   mySpecification =
            string("struct                                    ") +
            string("{                                         ") +
            string("    struct                                ") +
            string("    {                                     ") +
            string("          (3)field11;                     ") +
            string("          (2)field12;                     ") +
            string("    }field1[BLOCKSIZE/5]                  ") +
            string("} buffer;                                 ");

char  sourceBuf[BLOCKSIZE];

BufferManagementClass
      BufferWrapper(sourceBuf, sizeof(sourceBuf),mySpecification);
```

Figure 3.11
Use of BLOCKSIZE keyword to specify the number of elements in an array

3.1.7) Flexible B-tree Declarations

The layout descriptor shown in Figure 3.12 reserves a minimum of 2 2-byte elements for field1, and two more, if space is available. Also the BLOCKSIZE keyword may be used to request LADEL to allocate as many records as will fit in the available storage. The layout descriptor shown in Figure 3.13 requests that a minimum of 1 3-byte element for field1 be reserved and BLOCKSIZE/3 elements more if space is available. This feature of LADEL can be used in implementing B-trees where it is ideal to fit as many records in the available storage as possible.

```
string   mySpecification =
            string("struct                                    ") +
            string("{                                         ") +
            string("    (1)field11;                           ") +
            string("    (1)field12;                           ") +
            string("}field1[2,2];                             ");

char  sourceBuf[BUFFER_SIZE];

BufferManagementClass
      BufferWrapper(sourceBuf, sizeof(sourceBuf), mySpecification);
```

Figure 3.12
Use of second array qualifier to specify the maximum number of additional elements that should be added to the array.

3.1.8) The SURPLUS Keyword

The SURPLUS keyword may be used to denote the number of bytes in the underlying block buffer, beyond the minimum, that could be distributed at the current level of the field hierarchy.

```
string   mySpecification =
       string("struct                          ") +
       string("{                               ") +
       string("   (1)field11;                  ") +
       string("   (1)field12;                  ") +
       string("   (1)field13;                  ") +
       string("}field1[1,BLOCKSIZE];           ");

char  sourceBuf[BUFFER_SIZE];

BufferManagementClass
      BufferWrapper(sourceBuf, sizeof(sourceBuf), mySpecification);
```

Figure 3.13
Use of BLOCKSIZE keyword to allocate as many records as fit in available storage

The value of SURPLUS at the topmost level of a field hierarchy is equal to BLOCKSIZE, less the layout descriptor's minimum allocation. The value of SURPLUS at an inner level of the hierarchy is determined relative to the number of bytes obtained by that level's parent during descriptor evaluation.

The data layouts of Figure 3.14 and Figure 3.15 illustrate the use of the SURPLUS keyword to specify the allocation of surplus bytes in the underlying buffer.

```
string   mySpecification =
       string("struct                                ") +
       string("{                                     ") +
       string("   (2, SURPLUS/2) field1;             ") +
       string("   (4, SURPLUS/2) field2;             ") +
       string("}block;                               ");

char  sourceBuf[128];

BufferManagementClass
      BufferWrapper(sourceBuf, sizeof(sourceBuf), mySpecification);
```

Figure 3.14
Use of SURPLUS keyword to allocate surplus bytes

The layout descriptor shown in Figure 3.14 generates the following storage assignments, relative to an n-byte-long buffer and m surplus bytes:

-. “no assignment possible" exception (n <= 5).

-. struct { (2+m) field1; (4+m) field2; } block; (n = 2m+6, m>=0);

-. struct { (3+m) field1; (4+m) field2; } block; (n = 2m+7, m>=0);

The layout descriptor shown in Figure 3.15 generates the following storage assignments, relative to an n-byte-long buffer and m surplus bytes:

-. “no assignment possible" exception (n <= 2).

-. struct {struct {(1) subf1; (1) subf2; } field1; (n-2) field2; } block; (3 ≤ n ≤ 8);

-. struct {struct {(m) subf1; (m) subf2; } field1; (7) field2; } block; (n = 2m+7, m ≥ 1);

-. struct {(2m+1) struct{(m) subf1; (m) subf2;}field1; (7) field2;}block; (n=2m+8, m≥1);

```
string   mySpecification =
       string(“struct                                          ”) +
       string(“{                                               ”) +
       string(“   (0, SURPLUS)                                 ”) +
       string(“   struct                                       ”) +
       string(“   {                                            ”) +
       string(“         (1, SURPLUS/2)field11;                 ”) +
       string(“         (1, SURPLUS/2) field12;                ”) +
       string(“   }field1;                                     ”)+
       string(“   (1, 6) field2;                               ”)+
       string(“}block;                                         ”);

char  sourceBuf[128];

BufferManagementClass
      BufferWrapper(sourceBuf, sizeof(sourceBuf), mySpecification);
```

Figure 3.15
Another example of use of SURPLUS keyword to allocate surplus bytes

Storage expressions that evaluate to negative values (e.g., SURPLUS-6, SURPLUS ≤ 5) are treated as "0" for the purpose of storage distribution.

The SURPLUS and BLOCKSIZE fields may not be used together in a single specification. Attempting to do so will cause the BufferManagementClass constructor to throw an InvalidUseofKeyWordException exception.

3.2) LADEL's Formal Grammar

The following is the grammar for strings that describe block layouts:

fieldSpec	::=	sizeQualifier fieldPart vectorQualifier ;
sizeQualifier	::=	(sizeExpr)
	::=	(sizeExpr, surplusSizeExpr)
	::=	NULL
fieldPart	::=	struct { fieldSpecList }
	::=	*fieldName* (i.e., an alphanumeric identifier)
	::=	NULL
vectorQualifier	::=	[sizeExpr, sizeExpr]
	::=	[sizeExpr]
	::=	NULL
sizeExpr	::=	sizeTerm
	::=	sizeExpr ± sizeExpr

sizeTerm	::=	sizeFactor
	::=	sizeTerm * sizeTerm
	::=	sizeTerm / sizeTerm
sizeFactor	::=	(sizeExpr)
	::=	naturalNumber
	::=	C++ baseType
surplusSizeExpr	::=	surplusSizeTerm
	::=	surplusSizeExpr ± flexibleSizeExpr
surplusSizeTerm	::=	surplusSizeFactor
	::=	sizeTerm * surplusSizeTerm
	::=	surplusSizeTerm * sizeTerm
	::=	surplusSizeTerm / sizeExpr
surplusSizeFactor	::=	(surplusSizeExpr)
	::=	*naturalNumber* (i.e., a nonnegative integer)
	::=	*C++ baseType* (i.e., char, unsigned char, int, etc.)
	::=	BLOCKSIZE
	::=	SURPLUS
fieldSpecList	::=	fieldSpec
	::=	fieldSpec fieldSpecList

3.3) LADEL Buffer Management Object Operators and Methods

Storage associated with LADEL buffer management objects may be manipulated using the <<, >>, ^, and [] operators:

- The extraction operator, >>, streams objects from a BMO-managed buffer, according to the type of the right-hand argument. An expression of the form foo >> baseVariable, where
    - foo is a is a storage descriptor whose type is same as baseVariable type, and
    - baseVariable is an object of type "built-in" i.e., a char, an int, a float, etc.,

  streams a sequence of bytes from foo into baseVariable. Every BMO is associated with a current position cursor that tracks the next byte to be written into or read from the BMO's associated buffer. The expression "foo >> baseVariable"
    - copies the next sizeof(baseVariable) bytes at the current position in foo into baseVariable, then
    - advances foo's current position indicator by sizeof(baseVariable) bytes.

- The insertion operator, <<, inserts objects into a BMO-managed buffer, in a way that is comparable to the operation of the extraction operator >>.

- The selection operator, ^, retrieves a child BMO from a BMO. An expression of the form (foo^"xxx") >> bar, where foo and bar denote BMOs and "xxx" names a top-level field in foo, copies the BMO for foo's xxx component into bar.

- The indexing operator, [], also retrieves a subordinate (child) BMO from a BMO. An expression of the form foo[n] >> bar, where foo and bar denote BMOs and n is the index of a top-level field in foo, copies the top-level hierarchical storage descriptor for foo's nth component into bar. (0-offset indexing is used).

A final BMO-based method, the positioning operator setpos(), resets the "current byte within field" index for streaming operations.

The selection and streaming operators raise exceptions when invalid operations are attempted: i.e., when an unknown or an anonymous field is selected, or when a streaming operator attempts to access locations beyond the boundaries of a sub field

3.4) LADEL's Buffer Management Object

A Buffer Management Object (BMO) manages a user specified block buffer of type char. A BMO also manages buffer data for the user. BMOs provide methods and operators for data access and manipulation using syntax similar to C++. A BMO user can create a specification for a data structure in a string format and pass this specification to a BMO object constructor. This initial BMO, referred to here as a Top Level BMO object, then parses the data layout string and creates a set of child BMOs. The children of the Top Level BMO object provide direct access to individual fields in the data layout.

The following document the usage of each individual method and operator of BMO. The explanations are given with reference to layout in Figure 3.16.

1. **Constructor**

   BufferManagementClass(char* myBuffer, int sizeOfBuffer, stringClass& dataLayout)

   - **Parameters**
     - myBuffer : A user specified data buffer that will hold the user's data. .
     - sizeOfBuffer: Size of the user specified buffer
     - dataLayout : A string object defining a structured view of myBuffer.

   - **Example**

     BufferManagementClass

     ToplevelBmo(myBuffer, sizeOfBuffer, mySpecification);

2. **Selection Operators**

   I. BufferManagementClass&

   BufferManagementClass::operator^(const stringClass& FieldName);

   - **Parameters**

- FieldName: Name of the field in the user specified data structure.

- **Return Value**

  Returns a reference to a BMO representing the field FieldName

- **Example**

  (ToplevelBmo^"Name") returns a reference to a BMO representing the field Name.

```
stringClass  mySpecification  =
            stringClass(" struct                              ") +
            stringClass(" {                                   ") +
            stringClass("     (char*10) Name;                 ") +
            stringClass("     (char) flag;                    ") +
            stringClass("     struct                          ") +
            stringClass("     {                               ") +
            stringClass("           struct                    ") +
            stringClass("           {                         ") +
            stringClass("             (int)PostBoxNo;         ") +
            stringClass("             (char*21)Street;        ") +
            stringClass("             (char*12)City;          ") +
            stringClass("           } Address; "              ") +
            stringClass("           } Addresses[4]; "         ") +
            stringClass("           double Salary; "          ") +
            stringClass("}Person                              ");

 int sizeOfBuffer = 10 * sizeof(char) + sizeof(char)
                  + 4 * (sizeof(int) + 21 * sizeof(char)
                  + 12 * sizeof(char) ) + sizeof(double);

char *myBuffer = new char[SizeOfBuffer];
```

Figure 3.16
Layout used for explanation of BMO

3. Insertion Operations

BufferManagementClass::operator << (const int& myInt);

- **Parameters**

  - myInt: integer to insert into the field represented by the BMO.

- **Example**

  int myInt = 23456;

  (((ToplevelBmo^"Addresses")[0])"PostBoxNo") << myInt;

The << operator is also overloaded for objects of 'float', 'double', 'char', stringClass, and null terminated strings.

4. Extraction Operations

BufferManagementClass::operator >> (int& myInt);

- **Parameters**
  - myInt : a variable of type integer. Data will be extracted from the field represented by the BMO object into myInt.
- **Example**

  int myInt;

  ((ToplevelBmo^"Addresses")[0])"PostBoxNo") >> myInt;

The << operator is also overloaded for objects of 'float', 'double', 'char', stringClass, and null terminated strings.

5. Miscellaneous

I. BufferManagementClass::setPos(int Pos);

- **Parameters**
  - Pos: a variable of type integer. The current position in the field represented by the BMO will be set to pos.

- **Example**

  (ToplevelBmo^"Salary").setpos(0);

II. int BufferManagementClass::getSize(void);

- **Return Value**

  Returns the size of the field represented by the BMO.

- **Example**

  (ToplevelBmo^"Name").getSize() will return 10;

III. BufferManagementClass::getNumberOfRecords();

- **Return Value**

  Returns the number of fields within an array of fields of same type.

- **Example**

  (ToplevelBmo^"Address").getNumberOfRecords will return 4

IV. stringClass BufferManagementClass::getBmoName();

- **Return Value**

  Returns a string containing the name of the BMO.

- **Example**

  (ToplevelBmo^"Salary").getBmoName will return "Salary";

  (ToplevelBmo^"Addresses")[0].getBmoName will return "Address[0]";

To summarize, LADEL provides C-like declarations for specifying buffer layout with support for nested structs with named, typed fields. It also supports dynamic buffer layout: i.e., the ability to distribute space in buffer that may be available at run time, beyond that buffer's minimum requirements to function. LADEL also provides operator-based buffer manipulation, with support for selection-operator-based field access and support for safe, stream-based manipulation of individual fields, including checks for overflow and underflow on a per-field basis. This support for buffer manipulation is provided with no language-specific extensions to C++. LADEL was implemented as a language within C++, using a class whose constructor processes a layout specification in string format.

CHAPTER 4

Readability and Performance Evaluation

This chapter consists of two sections. Section 4.1 evaluates the complexity and readability of code that manipulates block buffer data. Two sample buffer manipulation problems, involving a simple and a moderately complex data layout, are used as test cases to argue the claim that LADEL simplifies buffer management logic. The standard C++ manual buffer manipulation idioms yield complex, low-level codes, especially when compared to the code produced using LADEL. The manual buffer manipulation code for the moderately complex layout is also substantially more complex than the manual buffer manipulation code for the simple layout. This substantial increase in complexity contrasts with the LADEL examples, which exhibit a much smaller increase in size, relative to the complexity of the data layout.

Section 4.2 describes two series of tests that were conducted to determine the performance of LADEL. The first series of tests involved packing data for network transmission in a sender program, sending the data, and finally unpacking these data at a receiver. The time required to pack a buffer using classic C++ typecasting code was compared to the time required for buffer packing with LADEL code. These tests were conducted once with a simple data layout, and once with complex data layout. The second series of tests assessed the execution time required by LADEL's constructor. The tests suggest that LADEL's overhead is small, particularly when constructors and selection are invoked in judicious ways.

4.1) Complexity and Readability Evaluation

It has already been stated that manually packing the contents of struct into a char buffer, writing the buffer to a device, and then unpacking the buffer at the receiving end produces low-level, messy code. The two examples below demonstrate how the use of LADEL results in simpler and cleaner code that is easier to read and maintain.

4.1.1) Simple Example

Assume that the following information needs to be packed into a buffer, outgoingBuffer, prior to being written to a network socket: name of a person [e.g., "Smith"]; person's address, including post box number [e.g., 23456], street [e.g., "North Greenwood Drive"], and city [e.g., "Johnson City"]; and person's salary [e.g., 50000.00].

In the standard C++ idioms for buffer packing, the user first creates a structure of the required type (cf. Figure 4.1). The programmer then fills the structure, and packs the buffer using casting and indexing, as shown in Figure 4.2, or the memcpy() function, as shown Figure 4.3.

The sizeof() and casting, along with the need for byte-by-byte copying, make the code in Figure 4.2 long-winded and tedious to read and write. The byte-by-byte copying is needed to avoid alignment errors. The briefer code in Figure 4.3 is still cryptic, low-level and difficult to maintain.

```
struct  PersonType {
      char Name[10];
      struct {
                  int PostBoxNo;
                  char Street[21];
                  char City[12];
            }Address;
            double Salary;
          }


struct PersonType Person;

Person.Name = “Smith”;
Person.Address.PostBoxNo = 23456;
Person.Address.Street = “North Greenwood Drive”;
Person.Address.City = “Johnson City”;
Person.Salary = 50000.00;
```

Figure 4.1
C++ structure declaration and initialization for simple example

Figure 4.4 and 4.5 show how the same task is handled using LADEL. The data structure is created as a string and passed as an argument to the constructor of BufferManagementClass, which manages the block buffer data on the programmer’s behalf. The code in Figure 4.5 demonstrates that the code required to pack a buffer with data using LADEL is much simpler than manual packing code. Also the code is easy to read and maintain. Note, in particular, the absence of sizeof() operators and expressions that compute offsets into the buffer.

```
const unsigned outgoingBufferPayloadLength =
      sizeof(Person.Name) + sizeof(Person.Address.PostBoxNo) +
      sizeof(Person.Address.Street) + sizeof(Person.Address.City) +
      sizeof(Person.Salary);

const unsigned outgoingBufferTerminatorLength = sizeof(char);

const unsigned outgoingBufferLength =
           sizeof(outgoingBufferPayloadLength)
         + outgoingBufferPayloadLength
         + outgoingBufferTerminatorLength;

char *const pOutgoingBufferPayloadLength = &outgoingBuffer[0];

char *const Name =
      &outgoingBuffer[sizeof(outgoingBufferPayloadLength)];

char * const PostBoxNo = &outgoingBuffer[
          sizeof(outgoingBufferPayloadLength) + sizeof(Person.Name)];

char *const street =
      &outgoingBuffer[sizeof(outgoingBufferPayloadLength)
    + sizeof(Person.Name) + sizeof(Person.Address.PostBoxNo)];

char *const city = &outgoingBuffer[sizeof(outgoingBufferPayloadLength)
         + sizeof(Person.Name) + sizeof(employee->Address.PostBoxNo)
         + sizeof(Person.Address.Street)];

char *const salary =
      &outgoingBuffer[sizeof(outgoingBufferPayloadLength)
    + sizeof(Person.Name) + sizeof(Person.Address.PostBoxNo)
    + sizeof(Person.Address.Street)+ sizeof(Person.Address.City)];


for(unsigned i = 0; i < sizeof(outgoingBufferPayloadLength); i++)
     pOutgoingBufferPayloadLength[i] =
                         ((char *)(&outgoingBufferPayloadLength))[i];

for(i = 0; i < sizeof(employee->Name); i++) Name[i] = Person.Name[i];

for(i=0; i<sizeof(float); i++)
   Salary[i] = ((char *)(&(employee.Salary)))[i];

for(i=0; i<sizeof(int); i++)
   PostBoxNo[i] = ((char *)(&(Person.Address.PostBoxNo)))[i];

for(i = 0; i < sizeof(Person.Address.Street); i++)
           Street[i] = Person.Address.Street[i];

for(i = 0; i < sizeof(Person.Address.City); i++)
           City[i] = Person.Address.City[i];

outgoingBuffer_pTerminator = '\0';
```

Figure 4.2
Code demonstrating manual packing with casting and indexing for simple example

```
const unsigned outgoingBufferPayloadLength =
      sizeof(Person.Name) + sizeof(Person.Address.PostBoxNo)
    + sizeof(Person.Address.Street)
    + sizeof(Person.Address.City) + sizeof(Person.Salary);

const unsigned outgoingBufferTerminatorLength = sizeof(char);

const unsigned outgoingBufferLength =
      sizeof(outgoingBufferPayloadLength)
    + outgoingBufferPayloadLength  + outgoingBufferTerminatorLength;

char *const pOutgoingBufferPayloadLength = &outgoingBuffer[0];

char *const Name =
            &outgoingBuffer[sizeof(outgoingBufferPayloadLength)];

char * const PostBoxNo = &outgoingBuffer[
          sizeof(outgoingBufferPayloadLength) + sizeof(Person.Name)];

char *const street =
            &outgoingBuffer[sizeof(outgoingBufferPayloadLength)
          + sizeof(Person.Name) + sizeof(Person.Address.PostBoxNo)];

char *const city = &outgoingBuffer[sizeof(outgoingBufferPayloadLength)
          + sizeof(Person.Name) + sizeof(employee->Address.PostBoxNo)
          + sizeof(Person.Address.Street)];

char *const salary =
            &outgoingBuffer[sizeof(outgoingBufferPayloadLength)
          + sizeof(Person.Name) + sizeof(Person.Address.PostBoxNo)
          + sizeof(Person.Address.Street)
          + sizeof(Person.Address.City)];

memcpy(pOutgoingBufferPayloadLength,
            ((char *)(&outgoingBufferPayloadLength)),
                  sizeof(outgoingBufferPayloadLength));

memcpy(Name, Person.Name, sizeof(employee->Name);

memcpy(Salary, ((char *)(&(employee.Salary))), sizeof(float));

memcpy(PostBoxNo, ((char *)(&(Person.Address.PostBoxNo))),
                                              sizeof(int));

memcpy(Street, Person.Address.Street, sizeof(Person.Address.Street));

memcpy(City, Person.Address.City, sizeof(Person.Address.City));

outgoingBuffer_pTerminator = '\0';
```

Figure 4.3
Code demonstrating manual packing with the use of memcpy for simple example

```
stringClass mySpecification  =
                  stringClass(“struct                              ”) +
                  stringClass(“{                                   ”) +
                  stringClass(“     (char*10) Name;                ”) +
                  stringClass(“     struct                         “) +
                  stringClass(“     {                              “) +
                  stringClass(“      (int)PostBoxNo;               “) +
                  stringClass(“      (char*21)Street;              “) +
                  stringClass(“      (char*12)City;                “) +
                  stringClass(“     } Address;                     “) +
                  stringClass(“     double Salary;                 “) +
                  stringClass(“}Person                              “);
```

Figure 4.4
LADEL structure declaration for simple example

```
int SizeOfBuffer = 10 * sizeof(char) + sizeof(int) + 21 * sizeof(char)
                  + 12 * sizeof(char) + sizeof(double);

char *outgoingBuffer= new char[SizeOfBuffer];

BufferManagementClass
      outbufBmo (outgoingBuffer, SizeOfBuffer mySpecification);

(outbufBmo ^"Name") << "Burlington";

((outbufBmo ^"Address")^"PostBoxNo") << 23456;

((outbufBmo ^"Address")^"Street") << "North Greenwood Drive";

((outbufBmo ^"Address")^"City")  << "Johnson City";

(outbufBmo ^"Salary") << double (5000.0);
```

Figure 4.5
LADEL structure data initialization and packing code for simple example

A third strategy that is sometimes used for buffer packing in C++ passes a struct to an I/O routine by casting the entire struct as objects of type char * [Uckan]. The resulting code, which

is shown in Figure 4.6, looks clean. This code, however, assumes that casting a struct as an object of type char * yields a consistent result. Unfortunately, the C++ language standard makes no guarantees whatsoever about how the individual components of classes and structs are to be positioned in physical memory. This idiom, accordingly, is potentially nonportable at best, and unsafe at worst.

```
struct PersonType Person;

Person.Name = “Smith”;
Person.Address.PostBoxNo = 23456;
Person.Address.Street = “North Greenwood Drive”;
Person.Address.City = “Johnson City”;
Person.Salary = 50000.00;


char *outgoingBuffer = (char *) (&Person);
```

Figure 4.6
Demonstrating blind casting of struct to char *.

In short, an informal comparison of the approaches shown above shows that LADEL provides a cleaner interface for block buffer management, while guaranteeing that data are laid out contiguously in the memory, in a deterministic way.

4.1.2) Complex Example

This second, more complex, example shows the use of LADEL to insert 10 records into a buffer. The records to be inserted have the following information:

- Last Name of the person

- First Name of the person
- 4 addresses of the person
  - Post Box Number
  - Street
  - City
- 4 Activities related to this person.

To hold this information the structure shown in Figure 4.7 needs to be defined. For the purpose of this example the same info will be inserted into all 10 employee records as shown in Figure 4.8

The code in Figures 4.9 demonstrates the use of a standard C++ idiom to pack the specified buffer. After the code in Figures 4.9 is executed, outgoingBuffer contains the data to be written to a socket.

```
struct employeeType
                  {
                        char Last_Name[7];
                        char First_Name[6];
                        struct
                        {
                              int PostBoxNo;
                              char street[22];
                              char city[13];
                        }Address[4];
                        struct
                        {
                              char type[5];
                              char description[24];
                        }Activity[5];
                  };

      employeeType employee[10];
```

Figure 4.7
C++ Structure declaration for complex example

```
for(unsigned i=0;i<10;i++)
      {
            employee [i].Last_Name = "Jordan";

            employee [i].First_Name = "Smith";

            for (int j=0; j<4; j++)
            {
                  employee [i].Address[j].PostBoxNo = 23456;

                  employee [i].Address[j].street =
                                          "North Greenwood Drive";

                  employee [i].Address[j].city = "Johnson City";

            }
            for(int k=0; k<5; k++)
            {
                  employee [i].Activity[k].type = "Call";

                  employee [i].Activity[k].description =
                                          "Send Mailer to Customer";
            }

      }
```

Figure 4.8
C++ structure data initialization for complex example

Figures 4.10 and 4.11 show how the same task is handled using LADEL. The data structure is created in string format and passed on to the constructor of BufferManagementClass. BufferManagementClass will manage the block buffer data on the programmer's behalf. After the code in Figure 4.11 is executed, outgoingBuffer has the data to be written to a socket.

```
const unsigned sizeOfOneRecord = sizeof(employee[0].Last_Name)
            + sizeof (employee[0].First_Name)
            + 4 * (sizeof(employee[0].Address[0].PostBoxNo)
            + sizeof (employee[0].Address[0].street)
            + sizeof (employee[0].Address[0].city))
            + 5 * (sizeof (employee[0].Activity[0].type)
            + sizeof (employee[0].Activity[0].description));

const unsigned outgoingBufferPayloadLength = 10 * sizeOfOneRecord;

const unsigned outgoingBufferLength =
      sizeof(outgoingBufferPayloadLength)
    + outgoingBufferPayloadLength;

char *outgoingBuffer = new char[outgoingBufferLength];

char *const pOutgoingBufferPayloadLength = &outgoingBuffer[0];

for(i = 0; i < sizeof(outgoingBufferPayloadLength); i++)
            pOutgoingBufferPayloadLength[i] =
                  ((char *)(&outgoingBufferPayloadLength))[i];


unsigned sizeOfOneAddress = sizeof(employee[0].Address[0].PostBoxNo)
                        + sizeof (employee[0].Address[0].street)
                        + sizeof (employee[0].Address[0].city) ;

unsigned sizeOfOneActivity = sizeof (employee[0].Activity[0].type)
                        + sizeof (employee[0].Activity[0].description) ;
```

Figure 4.9
Code demonstrating manual packing for complex example

```
for(i=0; i<10; i++)
      {
            char *const Last_Name = &outgoingBuffer[i * sizeOfOneRecord
                              +  sizeof(outgoingBufferPayloadLength)];

            for(int p = 0; p < sizeof(employee[i].Last_Name); p++)
                  Last_Name[p] = employee[i].Last_Name[p];

            char *const First_Name = &outgoingBuffer[i * sizeOfOneRecord
                              + sizeof (employee[i].Last_Name)
                              + sizeof(outgoingBufferPayloadLength)];


            for(p = 0; p < sizeof (employee[i].Last_Name); p++)
                  First_Name[p] = employee[i].First_Name[p];


            for(int j=0; j<4; j++)
            {
                  char *const PostBoxNo = &outgoingBuffer[
                                    i * sizeOfOneRecord
                                +    sizeof(employee[i].Last_Name)
                                +    sizeof (employee[i].First_Name)
                                +    j * sizeOfOneAddress
                                +    sizeof(outgoingBufferPayloadLength)];

                  for(p=0; p<sizeof(employee[i].Address[j].PostBoxNo);
                                                                  p++)
                    PostBoxNo[p]=
                    ((char *)(&(employee[i].Address[j].PostBoxNo)))[p];


                  char *const street = &outgoingBuffer[
                                i * sizeOfOneRecord
                              + sizeof (employee[i].Last_Name)
                              + sizeof (employee[i].First_Name)
                              + j * sizeOfOneAddress
                              + sizeof(employee[i].Address[j].PostBoxNo)
                              + sizeof(outgoingBufferPayloadLength)];


                  for(p = 0; p < sizeof (employee[i].Address[j].street);
                                                                  p++)
                        street[p] = employee[i].Address[j].street[p];
```

Figure 4.9 (Continued)

```
            char *const city = &outgoingBuffer[i * sizeOfOneRecord
                              + sizeof(employee[i].Last_Name)
                              + sizeof (employee[i].First_Name)
                              + j * sizeOfOneAddress
                              + sizeof(employee[i].Address[j].PostBoxNo)
                              + sizeof (employee[i].Address[j].street)
                              + sizeof(outgoingBufferPayloadLength)];


            for(p = 0; p < sizeof(employee[i].Address[j].city); p++)
                        city[p] = employee[i].Address[j].city[p];

            }

            for(int k=0; k<5; k++)
            {
                  char *const type = &outgoingBuffer[i * sizeOfOneRecord
                         + sizeof (employee[i].Last_Name)
                         + sizeof (employee[i].First_Name)
                         + 4 * sizeOfOneAddress + k * sizeOfOneActivity
                         + sizeof(outgoingBufferPayloadLength)];

                  for(p = 0; p < sizeof(employee[i].Activity[k].type);
                                                              p++)
                        type[p] = employee[i].Activity[k].type[p];


                  char *const description = &outgoingBuffer[
                                 i * sizeOfOneRecord
                               + sizeof(employee[i].Last_Name) +
                               + sizeof (employee[i].First_Name)
                               + 4 * sizeOfOneAddress
                               + k * sizeOfOneActivity
                               + sizeof(employee[i].Activity[k].type)
                               + sizeof(outgoingBufferPayloadLength)];

                  for(p = 0; p <
                  sizeof(employee[i].Activity[k].description);
                                                        p++)
                        description[p] =
                              employee[i].Activity[k].description[p];

            }
```

Figure 4.9 (Continued)

Comparing the two code fragments in the second example we can say that it is much easier to use LADEL for managing structured data in a block buffer than managing it manually. The two examples also show that as complexity of data layout increases the complexity of the code that manually manages buffer data increases. With LADEL the complexity of buffer management code grows much less quickly.

```
stringClass mySpecification;

mySpecification =
      stringClass("struct                                          ") +
      stringClass("{                                               ") +
      stringClass("     (int) SizeOfOutgoingBufer;                 ") +
      stringClass("     struct                                     ") +
      stringClass("     {                                          ") +
      stringClass("           (char*6) Last_Name;                  ") +
      stringClass("           (char*5) First_Name;                 ") +
      stringClass("           struct                               ") +
      stringClass("           {                                    ") +
      stringClass("                 struct                         ") +
      stringClass("                 {                              ") +
      stringClass("                   (int)PostBoxNo;              ") +
      stringClass("                   (char*21)street;             ") +
      stringClass("                   (char*12)city;               ") +
      stringClass("                 }Address[4];                   ") +
      stringClass("           }Addresses;                          ") +
      stringClass("           struct                               ") +
      stringClass("           {                                    ") +
      stringClass("                 struct                         ") +
      stringClass("                 {                              ") +
      stringClass("                       (char*4)type;            ") +
      stringClass("                       (char*23)description;    ") +
      stringClass("                 }Activity[5];                  ") +
      stringClass("           }Activities;                         ") +
      stringClass("     }Person[10];                               ") +
      stringClass("}employee;                                      ");
```

Figure 4.10
LADEL structure declaration for complex example

```
int SizeOfBuffer = 10 * (6 * sizeof(char) + 5 * sizeof(char)
                  + 4 * (sizeof(int)
                  + 21 * sizeof(char) + 12 * sizeof(char))
                  + 5 * (4 * sizeof(char) + 23 * sizeof(char)))
                  + sizeof(int);

char *outgoingBuffer = new char[SizeOfBuffer];

BufferManagementClass
      outbufBmo (outgoingBuffer, SizeOfBuffer, mySpecification);

(outbufBmo^" SizeOfOutgoingBufer") << SizeOfBuffer;

for(int i=0; i < 10; i++)
      {
            char mychar[16];
            stringClass Person = "Person[" + itoa[i, mychar, 10) + "]";
            BufferManagementClass& PersonBmo = (outbufBmo^Person);

            (PersonBmo^"Last_Name") <<  "Jordan";
            (PersonBmo^"First_Name") <<  "Smith";

            for(int j=0; j<4; j++)
            {
                  BufferManagementClass & AddressBmo =
                                                (PersonBmo^"Addresses")[j];
                  (AddressBmo ^"PostBoxNo") << 23456;
                  (AddressBmo^"street") << "North Greenwood Drive";
                  (AddressBmo ^"city") << "Johnson City";
            }
            for(int k=0; k<5; k++)
            {
                  BufferManagementClass & ActivityBmo =
                                                (PersonBmo^"Activities")[k];
                  (ActivityBmo^"type") << "Call";
                  (ActivityBmo^"description") << "Sent mailer to customer";
            }


      }

const unsigned outgoingBufferLength = outbufBmo.getSize();

char *outgoingBuffer = new char[outgoingBufferLength];

outbufBmo >> outgoingBuffer;
```

Figure 4.11
LADEL structure data initialization and packing code for complex example

4.2) Performance Evaluation

Performance testing was done for the examples in Figures 4.1, 4.2, 4.4, 4.5 and Figures 4.7 through 4.14. The testing was performed using a pair of client-server programs that exchange messages via TCP. The performance testing was conducted on Windows 2000 professional operating system, with Pentium III processor with128 MB RAM, using Microsoft's Visual C++ 6.0 compiler.

4.2.1) Case 1-1: Performance Testing Using a Simple Data Layout

The first set of tests was conducted using a data layout consisting of name of a person; an address, which itself consisted of post box number, street, and city; and a salary. This information is as depicted in Figure 4.1 in section 4.1. These tests were conducted once with manually managed buffer packing code (refer to Figure 4.2 and Figure 4.3 in section 4.1) and once with LADEL managed buffer packing code (refer to Figure 4.5 in section 4.1) .The first case consisted of four types of tests as depicted in table 4.2.1.

The LADEL part of the packing test was done in two different ways. First the required selection operations were performed as a part of each insert cycle. Then the required selection operations were performed only once at the beginning of the test. The references to the selected BMOs were then saved and used to access the BMOs throughout the rest of the loop. The results of this first set of tests are depicted in table 4.2.1.

Table 4.2.1 Results of performance test for Simple Data Layout				
Method	Data Packed 500,000 times at client, sent once to the server	Data received once at server, Unpacked 500,000	Data Packed and sent 500,000 times at client	Data received and Unpacked 500,000 times at server
Casting	13 Sec	14 Sec	4,500 Sec	4,500 Sec
LADEL with repeated selection	25 Sec	26 Sec	4,571 Sec	4,578 Sec
LADEL with saving BMO Ptrs	15 Sec	16 Sec	4,550 Sec	4,550 Sec

4.2.2) Case 1-2: Performance Testing Using a Complex Data Layout

In the second case tests were conducted using a data layout consisting of the name of a person; 4 addresses for that person, each of which consisted of post box number, street, and city; and 4 activities associated with that person, which in turn consisted of type and a description (cf. Figure 4.7). This test was also conducted once with manually managed buffer packing code (cf. Figures 4.8 through 4.11) and once with LADEL managed buffer packing code (cf. Figures 4.12 through 4.14). This case consisted of two types of tests as depicted in table 4.2.2.

The LADEL packing test was again conducted in two ways. First, the selection operations were performed with each insert all through the test: i.e. 500,000 times. In the second test, the selection operations were performed only once. During the first pass the pointers to BufferManagementClass objects were saved. The saved pointers were then used for subsequent insertion operations. The results of the second case tests are depicted in table 4.2.2.

Table 4.2.2 Results of performance test for Complex Data Layout		
Method	Data Packed 500,000 times at client, sent once to the server	Data received once at server, Unpacked 500,000
Casting	226 Sec	226 Sec
LADEL without saving BufferManagementClass Ptrs	2,500 Sec	2,500 Sec
LADEL with saving BufferManagementClass Ptrs	250 Sec	250 Sec

4.2.3) Case 2: Performance Testing for the constructor (which performs the actual layout)

The BufferManagementClass constructor was invoked 1,000,000 in two separate tests: once with the data layout depicted in Figure 4.4 and once with the data layout depicted in Figure 4.2. This test was conducted to determine the performance of LADEL's BMO tree creation code. The results from this test are shown in table 4.2.3.

4.2.4) Analysis of the Performance Results

For programs that did insertions and did not repeatedly use the selection operations, the improvement in code quality obtained from LADEL is achieved at a cost of an 11% increase in execution time over programs that use casting to manipulate block data. For programs that did insertion as well as sent and received data over the network, the improvement is achieved at a cost of approximately 1.2% of the overall program execution time. The conclusion here is that

the use of LADEL incurs negligible performance degradation for networking programs and minor performance degradation for any other types of program.

Table 4.2.3
Results of performance test for just laying out the data

	Time in sec for Laying out 1,000,000 times	Time in sec for Laying out 1 time
Specification of Data Layout in Figure 4.4 (BMO tree with 7 nodes created)	3,400 sec	3.4 Milliseconds
Specification of Data Layout in Figure 4.12 (BMO tree with 261 nodes created)	2,0000 sec	20 Milliseconds

The conclusion from the test conducted in section 4.2.3 is that LADEL's data layout code performs well particularly when constructors are invoked in judicious ways. The constructor of BufferManagementClass takes a reasonable amount of time for laying out the BMO tree for a reasonable data layout specification. When the number of fields in data layout specification increases to a high number as in data layout of figure 4.12 the performance degrades as expected.

A second performance problem involved the selection operations. As the complexity and the hierarchy of the layout descriptor increased, the performance of LADEL took a big hit. This performance degradation resulted from the increasing number of function calls when the selection operations were performed for objects deeper and deeper within the Block Management Object hierarchy. For each indirection the number of function calls increased by one. This problem was overcome by performing the selection operation only once and then saving the BufferManagementClass pointers from these operations. For subsequent insertions these saved

pointers were used. This considerably improved performance. The performance degradation associated with LADEL's selection operation could be eliminated by incorporating hash tables into Top Level BufferManagementClass object. These hash tables would reference every subordinate BufferManagementClass object in the top-level object's hierarchy. An effect similar to hash table functionality is achieved in the performance tests by saving references to all the BMOs within the BMO tree. This saving of references to BMOs is done in the test programs, as the current implementation of LADEL does not have this hash table support.

CHAPTER 5

CONCLUSIONS

The starting point for this thesis was the need for a language that simplifies the task of block buffer manipulation. The primary result of the work on this problem was a language, LADEL that allows C++ programmers to specify buffer layouts in a high-level way. Section 5.1 summarizes the work on LADEL presented in Chapters 1 through 4. Section 5.2 discusses further improvements to the LADEL language that were not implemented as a part of this thesis. Section 5.3 concludes with observations on the significance of the work.

5.1) Summary of Work

We have seen that managing block buffer data manually resulted in a low level code that was difficult to read and maintain. LADEL reduced the complexity of the buffer packing code by providing a high-level interface for buffer manipulation. LADEL also provided a way for structuring the data within a block buffer.

We have also seen that C++'s I/O facilities did not provide a set of high-level language constructs that allow control over the precise positioning of data in physical memory. LADEL solved this problem by supporting flexible data layout that gave control to the programmers over how data are positioned in memory.

We discussed two proposed strategies for solving the buffer manipulation problem. One strategy was to type cast the whole stuct of data as char *. The problem with this strategy was that it

assumed that data within a struct would be laid out contiguously in memory. For systems that did not lay data contiguously in memory, this strategy would result in non-portable or incorrect code or both. LADEL solved this problem by guaranteeing that the data within a LADEL specification would be laid out contiguously in memory.

The second strategy was to make a C++ class streamable and overload extraction (>>) and insertion (<<) operators in such a way that data can be streamed in and out of the class's internal variables. One of the problems with this strategy was that it did not allow programmers to treat a buffer as a hierarchical object consisting of sub-blocks with the ability to individually and independently stream data in an out of these sub-blocks. LADEL solved this problem by providing a mechanism for specifying flexible data layout and augmented this mechanism with operators that allow data from sub-blocks to be streamed in and out individually and independently.

Another problem with the second strategy was that it did not allow programmers to dynamically specify the data layout. LADEL solved this problem by allowing the programmers to create their data layouts in string format at run time.

We did some readability and maintainability comparisons between code performing buffer manipulation using LADEL and code performing buffer manipulation manually. The outcome of these comparisons was that the use LADEL resulted in code that was easy to read and maintain.

Some performance tests were run to determine how LADEL performs as compared to code that manually managed the block buffer data. When these tests were conducted it was found that the performance of LADEL degraded as the number of selection operations increased. As selection was performed on deeper and deeper fields within a data layout, the number of function calls increased, which resulted in performance degradation. Doing the selection operation only once and saving the pointers for streaming operations for the next pass solved this performance problem.

The results from these performance tests demonstrated that improved readability and maintainability of LADEL was achieved at a cost of 11% degradation in performance compared to the code that manually managed block buffer data. This degradation reduced to 1.1 % when the cost of transferring the packed data was taken into account. So it is concluded that use of LADEL does not have much affect on the performance of programs using it.

5.2) Ideas for Further Improvements

Currently LADEL solves many problems associated with block buffer manipulation. It is quite flexible and performs well. There are at least two desirable features that could be added to LADEL. The first improves the flexibility by allowing multiple data layouts to be passed to the constructor of BufferManagementClass. The second addition is to maintain a hash table of pointers to all the BMOs within the top level BMO. These two features are discussed below.

5.2.1) Support for Multiple Declarations within a Structure Definition

In the current implementation of LADEL only one statement per buffer declaration specification is allowed. It does not have any symbol table capabilities. For example consider the layout shown in Figure 5.1. It should be possible to specify the layout of Figure 5.2, which is equivalent to layout of Figure 5.1.

The constructor in data layout of Figure 5.2 takes another argument that tells the number of data layouts being passed to the constructor of BufferManagementClass. The last data layout in the array of layouts would be the actual specification that needs to be laid out on the underlying buffer. The BufferWrapper object then creates a symbol table where it will maintain all the declarations specified in first n-1 layouts specified in the array where n equal the size of the data

```
stringClass mySpecification =
        stringClass("struct                                    ") +
        stringClass("  {                                       ") +
        stringClass("       (char*10) Name;                    ") +
        stringClass("       struct                             ") +
        stringClass("       {                                  ") +
        stringClass("           struct                         ") +
        stringClass("           {                              ") +
        stringClass("                  (char*10) Street;       ") +
        stringClass("                  (char*21) City;         ") +
        stringClass("           } Address[4];                  ")+
        stringClass("            }Addresses                    ") +
        stringClass (" } Person;                               ");


char sourceBuf[256];


BufferManagementClass
      BufferWrapper(sourceBuf, sizeof(sourceBuf), mySpecification);
```

Figure 5.1
Data Specification Layout Supported Currently

layout array. This improvement to the LADEL will improve the readability of the specifications and allow the programmer to construct a big data layout as a collection of small data layouts.

```
stringClass mySpecifications[2];
stringClass mySpecification[0] =
        stringClass("typedef                  ") +
        stringClass("  struct                 ") +
        stringClass("  {                      ") +
        stringClass("    (char*10) Street;    ") +
        stringClass("   (char*21) City;       ") +
        stringClass("  }Address;              ");
stringClass mySpecifications[1] =
        stringClass("struct                   ") +
        stringClass("  {                      ") +
        stringClass("       (char*10) Name;   ") +
        stringClass("          Address[4];    ")+
        stringClass (" } Person;              ");

char sourceBuf[256];
 BufferManagementClass
      BufferWrapper(sourceBuf, sizeof(sourceBuf), mySpecifications, 2 );
```

Figure 5.2
Desired Data Specification Layout

5.2.2) Hash Table Creation in the Top Most BMO

In section 4.2 it was pointed out that the performance of LADEL degrades as the number of selection operation performed increases. This is due to increasing number of function calls that result as deeper and deeper fields are accessed within a layout. This problem could be solved by maintaining a hash table of all BMO pointers for the data layout in the top level BMO. So for example if the field City of the field Addresses[0] is to be accessed it should be possible to say

BufferWrapper^"Addresses[0]^City" instead of

(BufferWrapper^"Addresses[0]")^"City"

The first statement would require only one function call as compared to two for second statement. So the performance would sure improve with storing the BMO pointers at the top level BMO. But it is important to point out that we are seeing the performance problems when we perform about 4 million selection operations. So in cases where very few selection operations are performed LADEL's performance degradation will be negligible.

5.3) Conclusion

The aim of this thesis was to provide C++ programmers with a tool for simplifying the task of block buffer management. As demonstrated in Chapter 4, classic idioms for managing block buffer data produce complex and low-level code. This manually managed code is difficult to read and maintain. As the complexity of the data layout increases the readability and maintainability of standard C++ buffer management code decreased. But with the use of LADEL, the readability and maintainability improves, even with complex data layouts. LADEL also provides C++ programmers with ability to specify buffer layouts at run time. As discussed in Chapter 2 this ability to specify buffer layout at run time could be quite useful to applications that read their data layouts at run time from some file or a database. All these improvements were achieved without much degradation in performance as demonstrated in chapter 4.

REFERENCE LIST

Koenig , 2001. *http://www.cygnus.com/misc/wp/draft/* : “02-21-2001”

Microsoft, *2001 “msdn.microsoft.com/library/default.asp?URL=/library/specs/s1d202.htm”* : “02-21-2001”

Graham D. Parrington. *A Stub Generation System for C++*. Computing Systems 8 (2): 135-169 (1995)

Stroustrup. Posting, comp.lang.c++.moderated [Private communication, James Higgins to Prof. Pfeiffer (my advisor), September 1999 (?); exact date of original posting uncertain]

Uckan, Y. *Problem Solving Using C++* c. 1993, Times High Mirror Education Group, p. 742ff.

VITA

Ashfaq A. Jeelani

Personal Data:	Date of Birth: October 2, 1972
	Place of Birth: Hyderabad, India
	Marital Status: Single
Education:	Osmania University, Hyderabad, India;
	Computer Science, Bachelor of Engineering, 1994
	East Tennessee State University, Johnson City, Tennessee;
	Computer Science, Master of Science, 2001
Professional	
Experience:	Computer Specialist, Dept. of Housing,
	East Tennessee State University; Tennessee, 1997-1998
	Software Engineer, Siebel Systems; San Mateo
	California, 1999-current.